Using Multi-objective Design Space Exploration to Enable Run-time Resource Management for Reconfigurable Architectures

Giovanni Mariani*, Vlad-Mihai Sima [†], Gianluca Palermo[‡], Vittorio Zaccaria[‡], Cristina Silvano[‡] and Koen Bertels[†]

*ALaRI - University of Lugano, Lugano, Switzerland
Email: marianig@alari.ch
[†]Delft University of Technology, Delft, The Netherlands
Email: {v.m.sima, k.l.m.bertels}@tudelft.nl
[‡]Politecnico di Milano, Milano, Italy
Email: {gpalermo, zaccaria, silvano}@elet.polimi.it

Abstract—Resource run-time managers have been shown particularly effective for coordinating the usage of the hardware resources by multiple applications, eliminating the necessity of a full-blown operating system. For this reason, we expect that this technology will be increasingly adopted in emerging multi-application reconfigurable systems.

This paper introduces a fully automated design flow that exploits multi-objective design space exploration to enable run-time resource management for the Molen reconfigurable architecture. The entry point of the design flow is the application source code; our flow is able to heuristically determine a set of candidate hardware/software configurations of the application (i.e., *operating points*) that trade off the occupation of the reconfigurable fabric (in this case, an FPGA), the load of the master processor and the performance of the application itself. This information enables a run-time manager to exploit more efficiently the available system resources in the context of multiple applications.

We present the results of an experimental campaign where we applied the proposed design flow to two reference audio applications mapped on the Molen architecture. The analysis proved that the overhead of the design space exploration and operating points extraction with respect to the original Molen flow is within reasonable bounds since the final synthesis time still represents the major contribution. Besides, we have found that there is a high variance in terms of execution time speedup associated with the operating points of the application (characterized by a different usage of the FPGA) which can be exploited by the run-time manager to increase/decrease the quality of service of the application depending on the available resources[1].

I. INTRODUCTION

Reconfigurable systems represent as a suitable option to meet performance, power, and cost constraints that characterize the challenge of future supercomputing, provided that reconfiguration overhead is balanced with respect to computationally intensive workload [1]. In this context, commercial systems based on Field Programmable Gate Arrays (FPGAs) and standard multi-core CPUs have been already deployed to augment the capabilities of servers running performance-critical operations (see, for example, the Convey HC1 server [2]).

Given the mixed workloads and multi-application scenarios that characterize the future of reconfigurable systems, we speculate that a Run-time Resource Manager (RRM) software layer, either at the firmware level or at the operating system level, will be needed to dispatch reconfigurable resources to the applications (with suitable virtualization primitives) just as in purely microprocessor-based eco-systems [3], [4]. Current approaches to run-time resource management require that the application is written to expose a set of operating software configurations. Each of those configurations provides a different trade-off in terms of application performance, power consumption and resource usage. In the reconfigurable computing scenario considered in this paper, different configurations consist of different mappings of the application tasks on the heterogeneous (hardware and software) processing elements.

In this paper, we introduce a fully automated design flow that exploits multi-objective design space exploration (DSE) to enable run-time resource management for the Molen reconfigurable architecture [5]. Starting from the application source code, our flow extends Molen to heuristically determine a set of candidate hardware/software operating points that trade off the occupation of the reconfigurable FPGA fabric, the load of the master processor and the performance of the application itself. To this end, we developed a DSE technique that identifies the most promising operating points by using profiling information coming from both software simulation and hardware synthesis.

The main contributions of this paper are the following:

- We introduce one of the first fully automated approaches to enable run-time resource management of multi-application reconfigurable systems.
- We analyze the overhead of the design space exploration and operating points extraction with respect to conventional reconfigurable synthesis flow such as Molen.
- We characterize the statistical behavior of the operating points for two reference audio applications both in terms of quantity (which may impact the speed of the run-time selection routine) and quality (how much the system speed can change when changing operating point).

The rest of the paper is organized as follows. Section II introduces an analysis of the background work. Section III introduces the target architecture and the proposed DSE framework while Section IV describes two case studies associated with two reference audio synthesis applications. Finally, in Section V we draw our conclusions and summarize the relevant contributions of this work.

II. BACKGROUND

Reconfigurable hardware technology is a consolidated set of methodologies, techniques and tools whose aim is to combine the flexibility of software and the speed and efficiency of hardware. In recent years, reconfigurable arrays (either *coarse-grained* - CGRAs or field-programmable-gate-arrays - FPGAs) have been either combined with VLIW processors [6], or to enhance the instruction set architecture of conventional CPUs [5], [7]. Deploying a reconfigurable application means partitioning it into a set of tasks, allocating them on the (heterogeneous) processing elements and scheduling their execution order. Since parallelism may be a very significant factor for improving resource usage efficiency and application speed, a great deal of research has been dedicated to devise techniques for automatic [8], or semi-automatic [9] parallelization. Parallelization and partitioning benefit either from independent mapping and scheduling [10], or from joint mapping and scheduling [11], [12]. Once application tasks are appropriately mapped, compilation and synthesis phases generate binaries for the master processor and bit-streams for the configurable devices [13].

All the previous approaches assume that the target platform is dedicated to execute a single application. Application partitioning and task mapping are thus done for each application considering that all computing resources are available. To the best of our knowledge, there has been no attempt in the past to devise techniques for mapping and scheduling re-configurable applications in a multi-application scenario. Our assumption is that the rate of adoption of reconfigurable systems in general computing and supercomputing will create a need to manage this increasingly complex scenario. In this paper we propose to tackle this problem by augmenting existing reconfigurable synthesis flow with design space exploration.

III. THE PROPOSED METHODOLOGY

The proposed methodology targets the design-time identification of different operating configurations for a target application. Once identified, the operating points are exposed to the Run-time Resource Manager (RRM) system software. In a multi-application scenario, the RRM is in charge to control the access to the configurable resources of the system. Since they are in limited number, the RRM should decide how to partition them among the running applications. The operating points that have been derived for each application are used for this purpose. Our main assumption is that the figures of merit that have been measured independently for each application do not vary significantly when the same applications run in a multi-application environment (i.e., *orthogonality* holds).

In the original Molen reconfigurable platform, each application is in charge of setting up its own reconfigurable resources (e.g. FPGA slots) through a **set** instruction and launch its execution through a **execute** instruction. The assumption is that
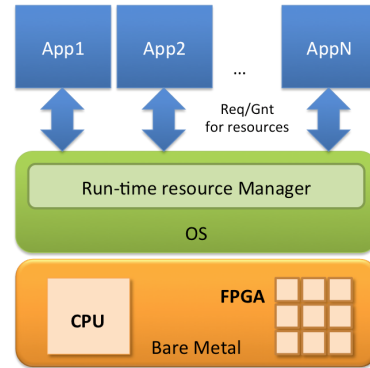

Fig. 1.   Structure of the hardware/software system at run-time.

the application has unconditional access to the reconfigurable resources.

In our extended Molen reconfigurable platform, each application should be granted access to the required resources by interacting with the RRM. Given the availability of operating points, the RRM can decide to partition and grant resources in order to maximize the overall, multi-application performance.

The considered RRM. In this paper, we work with a run-time resource manager that is aware of the performance that each application can reach, given a specific configuration of its hardware/software components. The performance (and other information) associated with each configuration is called *operating point* and will be described later in the paper. A list of operating points for each application is thus available to the run-time manager.

Each application has thus multiple operating points that are derived by trading off performance and resource usage. The run-time manager layer (which is aware of all the applications running simultaneously on the system, see Figure 1) can decide to change the current operating point of an application by means of an heuristic algorithm (in our case, we use a knapsack based heuristic [14]).

In our approach, we assume that the application, whenever it reaches a *Control Point* (CP), asks the run-time resource manager for how many reconfigurable resources it can use, given the current system-wide state. The CP is a point in the code where it is possible to choose between a hardware (reconfigurable) or software implementation (typically, a function call) of application tasks. Given that the system can have applications running on the reconfigurable device, the number of resources that the application is allowed to use at that particular moment may be less than in the stand-alone case.

In the following subsections, we will describe the original Molen platform and how we extended the original design flow to enable run-time resource management.

A. The Molen reconfigurable platform

The Molen architecture (Figure 2) is composed of a *master processor* and one or more blocks of *reconfigurable logic*. The software application runs on the master processor and can invoke the functionality of the configurable logic just as a co-processor.

Instruction fetch from the main memory is controlled by an arbiter. Conventional instructions that belong to the ISA of the
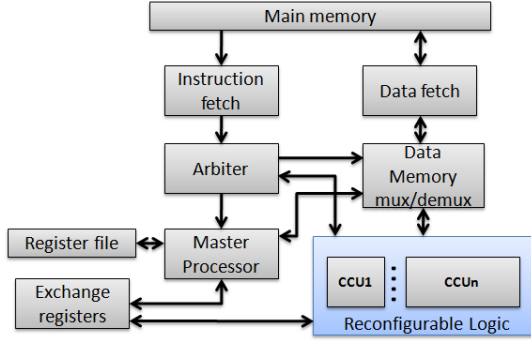
Fig. 2. *Molen* machine organization.

master processor are normally executed by the Microprocessor Unit itself. However, when a **set** instruction is decoded, the arbiter downloads the bit-stream of a specified Custom Computing Unit (CCU) onto the reconfigurable logic. In fact, a CCU is a hardware implementation of some application functionalities that has been synthesized beforehand.

When an **execute** instruction is decoded by the arbiter (provided that the CCU has been setup), the reconfigurable logic is started. Of course, as in co-processor based systems, the functionality of the CCU may need input parameters from the application; these parameters are passed through a BRAM memory that contains a set of exchange registers. In fact, whenever the CCU needs to read/write data from/to the main memory, the required memory block is first copied to the BRAM memory and then copied back to the main memory once the CCU execution is completed.

In the Molen architecture, the **set** instruction is typically scheduled in advance with respect to the **execute** instruction to hide the reconfiguration overheads.

The current Molen physical implementation is based on a Xilinx ML510 board. The master processor is the PowerPC (PPC) embedded in the Virtex5 FPGA, while the rest of the FPGA is used to store the reconfigurable CCUs, and basic system components, such as arbiter and memory controllers. Overall, the FPGAs available for the CCUs is 38400 LUTs while 5×64KB BRAM blocks are used for exchange registers. BRAM blocks are not shared between different CCUs; thus, in order to have homogeneous independent FPGAs slots available for CCUs, each slot is composed of maximum 7680 LUTs and 1 BRAM block to accommodate 1 CCU. If a CCU needs more than 7680 LUTs, it will occupy additional slots (thus reducing the total number of CCUs simultaneously stored).

B. The proposed design flow

An application can be configured in different ways to map some of its tasks on the *master processor* and some others on the FPGA slots. In the Molen design flow, a task corresponds to a C function (and all the callee functions of its call sub-tree). So, when an application invokes a C function that has been translated to CCU, it synchronously waits for its completion. In this case, we assume that the master processor can switch to another application (if it exists).

In our multiple-objective flow, each application configuration is characterized by a cost tuple:

$$\kappa = \langle \phi, \mu, \delta \rangle \qquad (1)$$

where ϕ is the number of FPGA slots occupied by the CCUs of the application, μ is the number execution cycles needed by the *master processor* to execute the application (not accounting for synchronous waits) and δ is the overall execution time of the application (from start to end).

The proposed design flow takes as input the C source code of the application and generates the application implementation in executable format (Figure 3). During the design process, a design space exploration phase is introduced to iteratively explore the space of the possible function-to-CCU mappings with the goal of minimizing the cost tuple κ. In fact, there might be different choices of mapping a function either to software (standard case) or to a CCU in hardware. In the latter case, the function and all the callee functions of its call sub-tree are mapped in hardware. Depending on the various choices, different operating points κ will be generated.
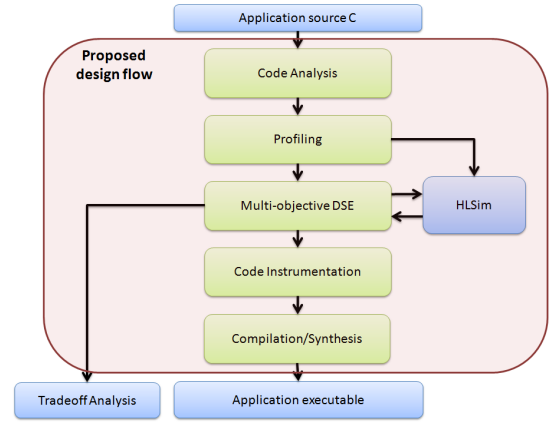

Fig. 3. The proposed design flow.

Code analysis. In this paper we use the same tool-chain front-end as the Molen architecture [13]. The C code is parsed and converted to an intermediate representation (IR) by using *Harmonic* and the *Rose compiler* [15], [16]. The application is then analyzed to identify which function can be converted to a CCU. Functions that present either direct or indirect recursion, as well as system and library call invocations, are automatically excluded from the conversion.

Profiling. The execution time of every function that may run on the master processor is profiled by executing the function directly on actual hardware (i.e., the PPC available on the ML510 board). Concerning the functions which can also be mapped on FPGAs, we profile both the execution time and the area occupation. The execution cycles are estimated by using *Modelsim* simulations of the actual VHDL code generated by the *Dwarv* high-level-synthesis compiler [17]. The area occupation is estimated by means of the analytical model presented in [18].

For the sake of the exploration, a *High Level Simulator* (*HLSim*) is automatically generated to be used as an executable model of the target platform. It takes as input the mapping specification of functions onto either master processor or FPGAs and it computes the execution time of the application based on the profiling information collected before. To trade-off accuracy and simulation speed, this is done by using dynamic profiling of the software program, i.e., the simulator

runs a pure software version of the application to measure how many times each function is invoked for a specific input data-set.

Multi-objective DSE. We integrated the multi-objective DSE tool *Multicube Explorer* [19] in the tool-chain. The tool identifies the optimal set of application mappings χ that provide the best trade-off for the cost tuple $\kappa(\chi)$.

An application configuration χ is a vector of n elements, each one representing a mapping configuration of a specific C function of the application. In our case, each element associated with a function can have 6 different levels (i.e. $PPC, fpga_1, ..., fpga_5$). In the case two functions are assigned to the same FPGA slots, a reconfiguration overhead incurs and the execution time of the application increases.

Optimization is done by using the Greedy Evolutionary Multi-objective Optimization (GEMO) algorithm [20]. The algorithm starts from a baseline configuration χ and iteratively maps functions to FPGA slots by trying to reduce application execution time δ (estimated with HLSim) and the usage of the master processor μ. The latter is done in order to unload as much as possible the master processor that can then be used for other processes (we assume a common time-sharing OS). The algorithm is iterated a number of times proportional to the quantity of free parameters (number of functions) and to the number of levels a parameter can assume (6).

The configurations belonging to the Pareto set of the multi-objective problem solved by GEMO are then enumerated by means of their FPGA resource requirement ϕ. If more configurations are found with the same cost ϕ, only the one with inferior δ is kept. ϕ is then used as configuration identifier by RRM.

Code instrumentation. In the presence of multiple operating-points, the application code is enhanced by adding control points (CPs) before the invocation of each function that has been chosen to be mapped on the FPGA for at least one configuration. The insertion of CPs consist in glue code to enable the RRM-supervised selection between hardware and software implementation of application tasks. At these control points, the application is supposed to ask to the run-time manager for available hardware resources (in terms of usable number of slots ϕ). It will then use the **set** and **execute** instructions to invoke the hardware functionality.

Compilation/synthesis. Finally, the VHDL code for both profiling and synthesis is generated by *Dwarv* [17]. The code is passed to a commercial synthesis tool to generate the configuration bitstreams implementing the CCUs. The remaining parts of the application source code are compiled and packaged together with the bitstreams to generate a single executable binary [13].

IV. EXPERIMENTAL RESULTS

In this section, we present the results obtained by applying the proposed design flow to two reference audio applications mapped on the Molen architecture. The target applications are a Wave Field Synthesis (WFS) [21] and an advanced In Car Audio player (INCAR) [22], [23]. WFS renders 3D audio by using multiple loudspeakers in a teleconferencing room; INCAR processes digitally an audio signal supposed to be listened in a car in order to eliminate noise, equalize with respect to car cabin features and compensate loudspeaker positioning.
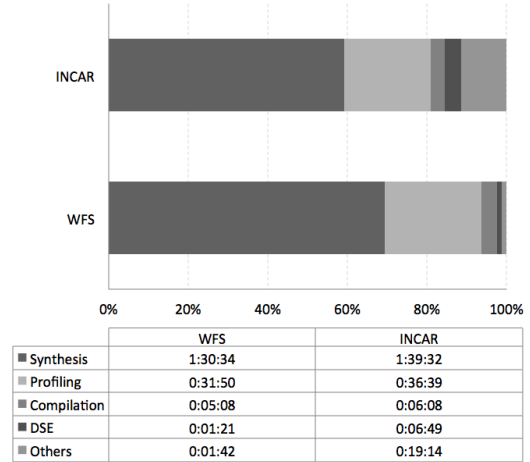

	WFS	INCAR
■ Synthesis	1:30:34	1:39:32
■ Profiling	0:31:50	0:36:39
■ Compilation	0:05:08	0:06:08
■ DSE	0:01:21	0:06:49
■ Others	0:01:42	0:19:14

Fig. 4. Design time breakdown for the proposed design flow considering the INCAR and WFS case studies.

Design flow analysis. We have run the proposed fully automated design flow for both WFS and INCAR and we measured the time required to perform each design. Overall the whole execution of the design flow took 2 hours and 11 minutes for the WFS application and, 2 hours and 49 minutes for the INCAR application.

The distribution of the time spent in the different design phases is shown in Figure 4. The most time consuming phase is still the synthesis step required to generate the bit-stream for the FPGAs. This phase alone takes about 2/3 of the overall time.

The second most important phase (in terms of time) is the profiling phase required to generate data for the high-level simulator *HLSim*. This phase takes from 22% to 25% of the overall time. The compilation time required to generate the binaries to run on the *master processor* is pretty short compared to the other design phases; in fact compilation takes 4% of the design time for both applications.

The remaining time is spent in the DSE phase and for other operations like the parsing of the source code and the generation of the intermediate representation. In particular, the DSE phase takes from 1% to 4% of the overall design time while other activities take from 1% to 11% of the design time. The low overhead introduced by the DSE time is due to the use of HLSim for evaluating the hardware/software configurations. Each evaluation took less than a second for each configuration, while presenting for both applications a measured average and maximum error of 4% and 9% respectively. The larger DSE and *Others* design-time associated with INCAR is likely due to the higher complexity of the application and to the significant number of signal processing steps that are considered to be mapped on hardware.

Trade-off analysis. Figure 5 reports the configurations found during the DSE phase for the target applications. We use bubble plots to characterize the application configurations in three dimensions: required FPGA slots (x-axis), speedup with respect to pure software (y-axis) and percentage usage of the master processor (bubble radius).

For the WFS case study, the DSE exposes three configurations (pure-software, 1 function on FPGA and 2 functions
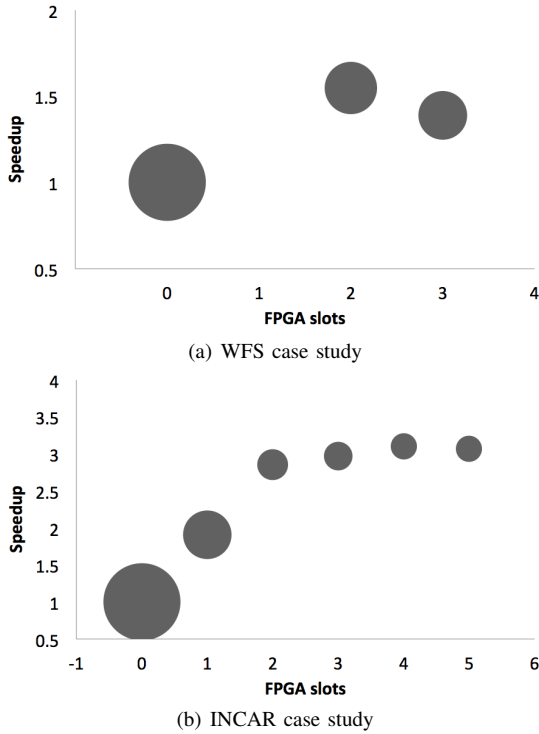
(a) WFS case study

(b) INCAR case study

Fig. 5. Trade off analysis for the INCAR and WFS case studies. The bubble plots report cost in terms of FPGA resource (x axis), cost in terms of master processor usage (bubble radius) and application speedup (y axis).

on FPGA). The first function to be mapped in hardware is too large to fit in a single FPGA slot, thus two FPGA slots should be allocated. At the cost of two FPGA slots we can achieve a speedup of 1.55. Mapping an additional function on the FPGA is possible. In this case, at the cost of three FPGA slots, a degradation of the performance is observed due to communication overheads (speedup is 1.39). This configuration is good since it is optimal in terms of *master processor* usage. The time required by the master processor to execute the WFS is respectively 4.59, 2.11 and 1.83 Giga cycles when using 0, 2 and 3 FPGA slots.

For the INCAR application, the DSE phase returns six different configurations that occupy from 0 to 5 FPGA slots. The maximum application speedup is 3.10 (mapping 4 functions on 4 different FPGA slots). Also in this case, it is possible to save a few *master processor* cycles by mapping an additional function on the fifth FPGA slot.

Comparison with a traditional design approach. In a traditional design approach, applications are optimized targeting the single-objective maximization of the performance [12]. The output of the design flow is a single application configuration, i.e. the one maximizing the performance.

On the other hand, the executable obtained with our approach should carry a software and hardware implementation for all the functions likely to be run on hardware. In addition, we must consider an execution overhead for switching to the most suitable configuration as selected by the RRM. Summing up, even if we have multiple configurations we still end up with a single executable with all the needed information.

Table I reports the size of binaries and configuration bit-streams, obtained with our approach (multi-objective) and with the traditional (single-objective).

Application	Approach	Binary size	Bit-stream size
WFS	Traditional	113356	952678
	Proposed	117682	1429017
	Overhead [%]	**3.9**	**50**
INCAR	Traditional	444438	1905356
	Proposed	450511	2381695
	Overhead [%]	**1.37**	**25**

We can observe that the overhead in terms of binary size is lower than 4%. However, the bit-stream overhead is considerable higher in our case (25% for INCAR and 50% for WFS) since there is the need to store all the CCU implementations required by the selected operating points.

In the proposed approach, we should also take into consideration the time overhead introduced by the additional instructions to check which configuration to run. This time overhead is anyway very low. In reference to the application execution time obtained with the traditional approach, the overhead introduced by the proposed approach is of 0.13% and 0.01% respectively for the two applications (without considering the time needed by the RRM to decide which configuration to use).

Run-time Resource Management Example. To show an example of run-time resource management enabled by the operating points found with our methodology, we run a synthetic workload where multiple instances of INCAR and WFS are launched randomly in time and each application instance terminates whenever it consumes a fixed input data set. Figure 6 shows the system behavior, both in terms of performance and resource usage for each application, when the following sequence of concurrent application scenarios (*snapshots*) is observed: INCAR only → INCAR and WFS → WFS and 2 instances of INCAR → 2 instances of INCAR. The objective of the RRM here is to maximize the application throughput while minimizing the resource usage (and meeting constraints on the available resources, both CPU and FPGA slots).

Besides, in Figure 6(a) we present the average normalized throughput with respect to the pure software implementation for each instance of the applications, while in Figure 6(b) and 6(c) we report, respectively, the number of FPGA slots and the CPU load for each application type. In the first snapshot, the presence of only a single instance of INCAR forces the RRM to select the *high-speed* operating point where four FPGA slots are used, since the operating point using all the FPGA slots presents a lower throughput (and a lower CPU usage). When WFS is launched (second snapshot), one of the FPGA slots is removed from INCAR (at the next control point) and assigned together with the free one to WFS, implying a reduction in INCAR throughput and an increase in CPU load. In the third snapshot, the RRM manages the arrival of the second instance of INCAR by: a) maintaining the same resource quantity for WFS, and b) reducing the FPGA slots given to the first INCAR instance. We can note, in this case, an additional pressure on the load of the CPU and an average throughput reduction for INCAR. Finally, when WFS consumes its input data stream and terminates, the two INCAR instances are assigned all the FPGA slots, thus increasing the average INCAR application throughput.

of the configuration will be rather small in terms of overhead.

(a) Applications Throughput
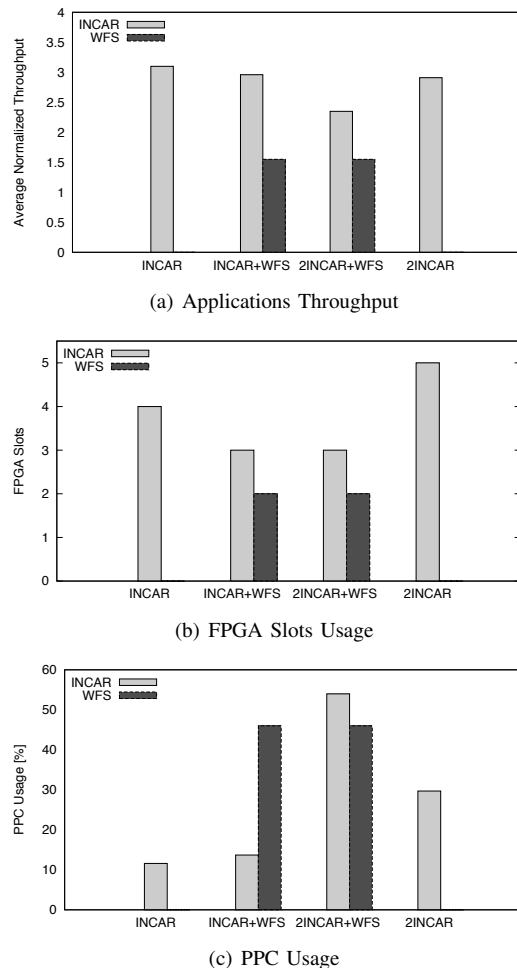

(b) FPGA Slots Usage

(c) PPC Usage

Fig. 6. System behavior in terms of (a) application throughput, (b) FPGA slots usage and (c) PPC usage, for 4 different snapshots of the synthetic workload composed by the following time sequence: INCAR only → INCAR and WFS → WFS and 2 instances of INCAR → 2 instances of INCAR

V. CONCLUSIONS

In this paper, we introduced a fully automated design flow that exploits multi-objective design space exploration to enable run-time resource management for the Molen reconfigurable architecture in a multi-application scenario.

We presented the experimental results where we applied the proposed design flow to two reference audio applications mapped on the Molen architecture. The analysis allowed us to conclude that the overhead of the design space exploration and operating points extraction with respect to the original Molen flow is within reasonable bounds in that the synthesis time still represents the major contribution (from 59% up to 69%). Besides, we have found that there is a high variance (up to 3×) in terms of execution time speedup associated with the operating points of the application, which can be exploited by the run-time manager to increase/decrease the quality of service of the application depending on the resources allocation. Finally we have found that the number of optimal (Pareto) operating points is rather small with respect to the overall number of possible configurations (from 3 up to 6). The latter feature allows us to speculate that the run-time selection

REFERENCES

[1] A. George, H. Lam, and G. Stitt. Novo-g: At the forefront of scalable reconfigurable supercomputing. *Computing in Science Engineering*, 13(1):82 –86, jan.-feb. 2011.
[2] T. M. Brewer. Instruction set innovations for the convey hc-1 computer. 30(2):70–79, 2010.
[3] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. *IET Computers & Digital Techniques*, 5(2):123–135, 2011.
[4] G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Arte: An application-specific run-time management framework for multi-core systems. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 86 –93, june 2011.
[5] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The molen polymorphic processor. 53(11):1363–1375, 2004.
[6] B. Mei, B. Sutter, T. Aa, M. Wouters, A. Kanstein, and S. Dupont. Implementation of a coarse-grained reconfigurable media processor for avc decoder. *Journal of Signal Processing Systems*, 51(3):225–243, 2008.
[7] G. Ansaloni, P. Bonzini, and L. Pozzi. Egra: A coarse grained reconfigurable architectural template. (99):1–13, 2010. Early Access.
[8] F. Ferrandi, L. Fossati, M. Lattuada, G. Palermo, D. Sciuto, and A. Tumeo. Automatic parallelization of sequential specifications for symmetric mpsocs. In *Proc. IESS07 - International Embedded Systems Symposium 2007*, pages 179–192, May 2007.
[9] J.-Y. Mignolet, R. Baert, T.J. Ashby, P. Avasare, Hye-On Jang, and Jae Cheol Son. Mpa: Parallelizing an application onto a multicore platform made easy. *Micro, IEEE*, 29(3):31 –39, may. 2009.
[10] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt. Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration. 14(11):1189–1202, 2006.
[11] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. 29(6):911–924, 2010.
[12] Y. M. Lam, J. G. F. Coutinho, and W. Luk. Integrated hardware/software codesign for heterogeneous computing systems. In *Proc. 4th Southern Conf. Programmable Logic*, pages 217–220, 2008.
[13] K. Bertels, V.-M. Sima, Y. Yankova, G. Kuzmanov, W. Luk, G. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti. HArtes: Hardware-software codesign for heterogeneous multicore platforms. 30(5):88–97, 2010.
[14] Ch. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal. Fast multidimension multichoice knapsack heuristic for mp-soc runtime management. *ACM Trans. Embed. Comput. Syst.*, 10:35:1–35:16, May 2011.
[15] W. Luk, J. G. F. Coutinho, T. Todman, Y. M. Lam, W. Osborne, K. W. Susanto, Q. Liu, and W. S. Wong. A high-level compilation toolchain for heterogeneous systems. In *Proc. IEEE Int. SOC Conf. SOCC 2009*, pages 9–18, 2009.
[16] Lawrence Livermore National Laboratory. Rose compiler, 2011. http://www.rosecompiler.org.
[17] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Yi Lu, and S. Vassiliadis. Dwarv: Delftworkbench automated reconfigurable vhdl generator. In *Proc. Int. Conf. Field Programmable Logic and Applications FPL 2007*, pages 697–701, 2007.
[18] R. J. Meeuws, C. Galuzzi, and K.L.M. Bertels. High level quantitative hardware prediction modeling using statistical methods. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Models, and Simulations*, pages 140–149, July 2011.
[19] Vittorio Zaccaria, Gianluca Palermo, and Giovanni Mariani. Multicube explorer, January 2008. http://www.multicube.eu.
[20] Marco Laumanns. *Analysis and Applications of Evolutionary Multiobjective Optimization Algorithms*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2003.
[21] K. Brandenburg, S. Brix, and T. Sporer. Wave field synthesis. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2009*, pages 1 –4, may 2009.
[22] J. Kontro, A. Koski, J. Sjoberg, and M. Vaananen. Digital car audio system. *Consumer Electronics, IEEE Transactions on*, 39(3):514 –521, jun 1993.
[23] K. Bertels, F. Bettarelli, S. Cecchi, E. Ciavattini, F. Ferrandi, W. Luk, F. Piazza, C. Pilato, A. Primavera, V. M. Sima, and R. Toppi. The hartes carlab: A new approach to advanced algorithms development for automotive audio. In *Audio Engineering Society Convention 129*, 11 2010.